Corrigé TP15

Ce document est accompagné des codes sources permettant de répondre aux questions de programmation.

Exercice 1

Le code source correspondant est Entrelacement.ml.

On constate qu'effectivement l'exécution du programme ne donne pas toujours les mêmes affichages. On observe néanmoins l'ordre partiel qui régit les instructions exécutées par un même fil : chacun d'entre eux affiche d'abord 10000000 puis 20000000 puis 30000000.

Exercice 2

1. Le code source correspondant est Compteur_faux.c.
2. Sans protection de l'incrément, le compteur contient en fin de chaque exécution une valeur différente qui n'est pas toujours 200000. Il peut être nécessaire de faire compter chacun des fils jusque plus que 100000 afin d'observer le phénomène : si le fil s'exécute rapidement, l'ordonnanceur pourrait faire le choix de ne pas entrelacer les deux fils, auquel cas, les incréments se font correctement.
3. Le code source correspondant est Compteur.c. Cette fois on obtient l'affichage attendu.

Exercice 3

1. On s'attendrait à avoir un fil 0 et un fil 1 qui comptent de 0 à 9 chacun. Il y a bien 20 affichages mais tous sont soit disant faits par le fil 1 : c'est incohérent.
2. On nous prévient à l'exécution d'une *data race*. Cette situation intervient lorsqu'au moins deux écritures ou une lecture et une écriture sur le même emplacement mémoire ne sont pas ordonnées. Parfois ça n'a pas d'incidence mais c'est généralement le signe d'un problème de concurrence.

   En l'occurrence, cette data race prend place dans la pile du fil appelant ce qui est anormal car il ne devrait pas y avoir de variable partagée par les fils appelés à cet endroit.
3. La durée de vie de la variable thread_index est limitée à une itération de la boucle, mais le fil créé par pthread_create reste vivant jusqu'au pthread_join et peut donc continuer d'utiliser le pointeur qu'on lui a fourni lors de sa création alors que la variable pointée n'existe plus.

   Dans une telle situation, le comportement du programme est indéfini. Ici, vu les affichages, il semblerait que : lors de l'itération 0, 0 est placé a un certain emplacement mémoire et lors de l'itération 1, cet emplacement mémoire est réutilisé pour stocker 1.
4. Il suffit de faire en sorte que l'objet vers lequel pointe l'argument passé au fil créé ait une durée de vie au moins égale à celle du fil. Le code source correspondant est vrai.c.

Exercice 4

Pour les questions 1 à 4, le code source correspondant est Petersen.c.

Pour la question 5, on constate que notre verrou ne fonctionne pas du tout comme attendu :

- Avec l'option -O0, c'est à dire sans optimisation, certaines exécutions n'atteignent pas 2000000. Il manque quelques incréments. Ceci n'est pas un problème de non atomicité mais de non respect de la *cohérence séquentielle* par le processeur. On ne creusera pas cette notion en cours.

- Avec l'option -O1, le programme ne termine plus. Ceci est du au fait que le compilateur risque de considérer dans la fonction lock que mutex->turn vaut évidemment other puisqu'on vient de le fixer à cette valeur et donc compiler lock comme si nous avions écrit :

```
void lock(petersen* mutex, int thread)
{
    mutex->want[thread] = true;
    int other = 1-thread;
    mutex->turn = other;
    while (mutex->want[other]) {}
}
```

  En conséquence, on obtient un interblocage.

- En compilant avec l'option -fsanitize=thread, on obtient effectivement des avertissements quant à des data race, ce qui est normal vu les deux points précédents.

La morale de l'histoire : un processeur et un compilateur ont en pratique le droit de faire des opérations que nous n'avons pas considérées lorsque nous avons prouvé la correction de l'algorithme de Petersen. Il n'y a donc pas de contradiction entre le fait que nous avons prouvé la correction de cet algrithme en cours et le fait qu'en pratique il ne fonctionne pas lorsqu'on l'implémente en C.

Exercice 5

Pour les questions 1 à 4, le code source correspondant est somme.c. Pour l'utiliser, il faut lancer l'exécutable avec deux arguments : le nombre de fils pour les versions parallèles et la taille du tableau dont on veut sommer les éléments. Vous ne verrez rien sur un tableau de taille trop petite. Les observations ci-après ont été faites avec un tableau de taille un million et 4 fils.

Pour la question 5 après quelques tests on constate que :

- La version parallèle qui utilise partial_sum_1 renvoie un résultat incorrect. C'est normal, pour les mêmes raisons qu'avec l'exemple simple où deux fils incrémentent un compteur partagé sans protection de l'incrément.

- Les versions parallèles avec partial_sum_2 et partial_sum_3 sont correctes. Pour l'observer, il suffit de constater que la différence entre la somme obtenue via ces fonctions et la somme obtenue via une boucle exécutée par un fil est extrêmement proche de 0.

  Remarque : On ne peut pas se permettre de tester l'égalité entre la somme obtenue de manière parallèle et celle obtenue naïvement : on aurait forcément une inégalité à cause du fait qu'on manipule des flottants. Rappel : tester une égalité entre flottants ne marche jamais.

- La version parallèle avec partial_sum_2 est significativement plus lente qu'avec un seul fil (environ cent fois plus). C'est normal, verrouiller et déverrouiller le mutex à répétition est coûteux.

- La version parallèle avec partial_sum_3 en revanche est significativement plus rapide qu'avec un seul fil (entre 5 et 10 fois plus rapide sur ma machine).

La morale de l'histoire : paralléliser une tâche peut considérablement réduire son temps de traitement... à condition de ne pas s'y prendre n'importe comment.

Exercice 6

Pour les questions 1 et 2, le code source correspondant est Producteur_consommateur.c.

Pour la question 3, on observe des affichages cohérents avec le fait que la taille du buffer vaut 5. Par exemple, on n'a jamais plus de 5 productions ou consommations d'affilée. On peut retracer l'évolution du contenu du buffer d'après les affichages provoqués par une exécution pour se convaincre que le buffer n'est jamais rempli au delà de 5 éléments et qu'aucun consommateur ne consomme alors que le buffer est vide : c'est le comportement recherché.

Exercice 7

Le code source correspondant est Producteur_consommateur_mutex.ml

1. Sans difficulté : il suffit de se rappeler comment initialiser une file vide et un mutex.
2. Pour ajouter dans le buffer sans problème de concurrence, le fil qui exécute produce commence par verrouiller l'accès au buffer puis fait son ajout avant de déverrouiller.
3. Pour la consommation, on fait de même : on verrouille l'accès au buffer puis on vérifie si on peut consommer. Si on peut, on le fait, sinon, on relâche le verrou de sorte à éventuellement permettre à un producteur de s'en saisir puis on recommence.
4. Le seul point un peu inhabituel est l'utilisation de Unix.sleepf. Cette fonction prend en entrée un flottant k et interrompt l'exécution du fil appelant pendant k secondes.
5. Pour des raisons de complexité, on fait les adjonctions en tête de liste puis on la renverse lors de l'affichage pour obtenir les éléments dans l'ordre dans lequel ils ont été consommés.
6. En surveillant la consommation de la machine, on constate que le processeur est utilisé à 100% par le programme pendant toute son exécution. C'est normal puisque le fil consommateur prend et relâche le verrou à répétition ce qui consomme des ressources qui seraient probablement plus utiles à faire autre chose !

   En pratique, ce programme pourrait même ne pas terminer : comme le fil consommateur monopolise le verrou, le court laps de temps pendant lequel il le relâche pourrait être insuffisant pour qu'un producteur s'en empare, entraînant une famine chez les fils producteurs.
7. Le code correspondant est Producteur_consommateur_sem.ml. La version 4.12 ou plus de Ocaml est nécessaire pour qu'il compile. Le principe est le même que dans l'exercice précédent, à ceci près que la taille du buffer n'est pas bornée donc on a uniquement besoin de gérer le cas où un consommateur cherche à consommer dans un buffer vide. On n'a donc besoin que d'un seul sémaphore.